Towards Ultra-scale Branch-and-Bound Using a High-productivity Language

Tiago Carneiro[a,*], Jan Gmys[c], Nouredine Melab[a,b], Daniel Tuyttens[c]

[a] *INRIA Lille - Nord Europe, France*
[b] *Université de Lille, CNRS/CRIStAL, France*
[c] *Mathematics and Operational Research Department (MARO), University of Mons, Belgium*

Abstract

Due to the highly irregular nature and prohibitive execution times of Branch-and-Bound (B&B) algorithms applied to combinatorial optimization problems (COPs), their parallelization has received these two last decades great attention. Indeed, significant efforts have been made to revisit the parallelization of B&B following the rapid evolution of high-performance computing technologies dealing with their associated scientific and technical challenges. However, these parallelization efforts have always been guided by the performance objective setting aside programming productivity. Nevertheless, this latter is crucial for designing ultra-scale algorithms able to harness modern supercomputers which are increasingly complex, including millions of processing cores and heterogeneous building-block devices. In this paper, we investigate the partitioned global address space (PGAS)-based approach using Chapel for the productivity-aware design and implementation of distributed B&B for solving large COPs. The proposed algorithms are intensively experimented using the Flow-shop scheduling problem as a test-case. The Chapel-based implementation is compared to its MPI+X-based traditionally used counterpart in terms of performance, scalability, and productivity. The results show that Chapel is much more expressive and up to 7.8× more productive than MPI+Pthreads. In addition, the Chapel-based

*Corresponding author
Email address: tiago-carnero.pessoa@inria.fr (Tiago Carneiro)

search presents performance equivalent to MPI+Pthreads for its best results on 1024 cores and reaches up to 84% of the linear speedup. However, there are cases where the built-in load balancing provided by Chapel cannot produce regular load among computer nodes. In such cases, the MPI-based search can be up to 4.2× faster and reaches speedups up to 3× higher than its Chapel-based counterpart. Thorough feedback on the experience is given, pointing out the strengths and limitations of the two opposite approaches (Chapel *vs.* MPI+X). To the best of our knowledge, the present study is pioneering within the context of exact parallel optimization.

Keywords: Ultra-scale optimization, Branch-and-Bound, PGAS, Chapel, MPI+X.

1. Introduction

Tree search algorithms are strategies that implicitly enumerate a solution space, dynamically building a tree. This class of algorithms is often used for the exact resolution of combinatorial optimization problems (COP), and it is present in many areas, such as operations research, artificial intelligence, bioinformatics, and machine learning [1, 2]. As COPs are often NP-hard, the size of problems that can be solved to optimality is limited, even if large-scale distributed computing is used [3, 4].

Among the tree search algorithms, the Branch-and-Bound (B&B) is one of the most widely used methods for solving instances of COPs to optimality. Due to its intrinsically parallel nature and prohibitive execution times [4], B&B algorithms have been revisited using several parallel computer architectures, including multicore [5], manycore processors [6, 7], and computational grids [8]. In this sense, it is expected that exascale computers will allow a significant decrease in the execution time required by B&B algorithms to solve COP instances to optimality. However, such large scale systems are going to be complex to program, and efforts towards productivity are crucial for better exploiting this future generation of computers [9, 10].

The parallelization efforts of B&B algorithms have always been guided by the performance objective setting aside productivity. Indeed, the focus is mostly put on the design of new data structures that are often problem-specific [11, 12, 13, 5]. Moreover, high-productivity languages historically suffer from severe performance penalties. Additionally, they often do not provide low-level features and are not suited to parallelism [14, 15]. Therefore, high-productivity languages are not commonly employed within the scope of parallel tree search. Instead, this kind of algorithm is frequently coded in C/C++, and different libraries and programming models are combined for exploiting parallelism and communication [3]. Among the high-productivity languages, Chapel is one designed for high-performance computing, and it is competitive to both C-OpenMP and MPI+X in terms of performance, considering different benchmarks [16].

The objective of the present research is to investigate whether it is possible to use a high-productivity language for efficient implementation of distributed B&B algorithms for solving COPs. To the best of our knowledge, the present research is the first one that investigates the use of a high-productivity language for this purpose. The primary challenge is to find a trade-off between productivity and performance, as parallel B&B algorithms often require hand-optimized code to achieve performance.

To accomplish the objective of this paper, we present a B&B algorithm conceived for the Chapel high-productivity language. This algorithm is implemented using the productivity-aware features of Chapel for distributed programming and applies both the global-view of control flow and data structures (PGAS) programming models, instead of the well-known Single Program - Multiple Data (SPMD). This implementation performs load balancing among different processes and also harnesses all CPU cores that a computer node has.

The experimental results show that, in the context of the present research, Chapel is almost 6× more expressive and from 2× to 7.8× more productive than MPI+Pthreads. The Chapel-based search presents performance equivalent to MPI+Pthreads for its best results on 32 computer nodes (1024 cores) and reaches up to 84% of the linear speedup. The productivity-aware features for

load balancing provided by Chapel are not enough for efficiently exploiting the computer resources of several locales in more irregular scenarios. In such cases, the MPI-based search can be up to 4.2× faster and reach speedups up to 3× higher than its Chapel-based counterpart.

The main contributions of this paper are the following:

- We present a parallel distributed B&B for solving permutation combinatorial optimization problems implemented using the productivity-aware features of Chapel for distributed programming. We intensively experiment the proposed algorithm using two lower bound functions, which result in two entirely distinct behaviors of the search.
- We analyze the influence of the distribution of the PGAS data structure on the overall performance of the implementation. Moreover, we study the effects of using atomic global view variables and the limits of the code automatically generated for exploiting the intra-node parallelism.
- Chapel provides three load balancing iterators for distributed programming. We also investigate which one is the best in the scope of B&B search algorithms.
- We compare the implementation in Chapel to a state-of-the-art B&B written in MPI+Pthreads. This comparison is made in terms of performance, scalability, and productivity.
- Finally, we discuss the results in terms of productivity, performance, and the road towards exascale. The insights provided by the present research may help potential users of other high-productivity languages and PGAS-based libraries, such as Unified Parallel C (UPC), X10, and OpenSHMEM.

The remainder of this paper is structured as follows. Section 2 brings background information and related works. Section 3 presents the distributed B&B algorithm written in Chapel. In turn, Section 4 presents a performance and scalability evaluation of the proposed implementation. A productivity-oriented

evaluation of Chapel for programming distributed B&B is performed in Section 5. Then, Section 6 brings a discussion of the results obtained in the present research. Finally, conclusions are outlined in Section 7.

2. Background and Related Works

2.1. The Chapel Programming Language

Chapel (Cascade High Productivity Language) is an open-source parallel programming language designed to improve productivity in high performance computing [17]. It incorporates features from compiled languages such as C, C++, and Fortran, as well as high-level concepts related to Python and Matlab. The parallelism is expressed in terms of lightweight tasks, which can run on several locales or a single one. In this work, the term *locale* refers to a symmetric multiprocessing computer in a parallel system [18].

In Chapel, both *global view of control flow* and *global view of data structures* are present [16]. Concerning the first one, the program is started with a single task, and parallelism is added through data or task-parallel features. Moreover, a task can refer to any variable lexically visible, whether this variable is placed in the same locale on which the task is running, or in the memory of another one. Regarding the second one, indexes of data structures are globally expressed, even in case the implementation of such data structures distributes them across several locales. Thus, Chapel is a language that applies the Partitioned Global Address Space (PGAS) programming model [19]. Finally, indexes of data structures are mapped to different locales using *distributions*. Contrasting to other PGAS-based languages, such as UPC and Fortran, Chapel also supports user-defined distributions [20].

Previous versions of Chapel were not yet a suitable replacement for C or Fortran+MPI in terms of performance. But, they could be instead a suitable replacement for both Matlab and Python [21, 22]. Chamberlain *et al.* [16] present the release 1.18 of the Chapel language, and show that it is competitive

to MPI+X, OpenMP, and SHMEM regarding performance, taking into account different benchmarks.

2.2. Tree Search Algorithms

Algorithms for solving combinatorial optimization problems can be divided into exact (complete) or approximate strategies [23]. Exact strategies guarantee to return an optimal solution for any instance of the problem in a finite amount of time. Complete algorithms for NP-hard problems usually apply concepts of enumerative strategies and therefore have exponential worst-case execution times [24]. In contrast, the approximate ones trade optimality against a good and valid solution obtained in reasonable time [25].

Tree search algorithms are strategies that implicitly enumerate a solution space, dynamically building a tree [2]. The internal nodes of the tree are partial solutions, whereas the leaves are complete solutions. Algorithms that belong to this class start with an initial node, which represents the root of the tree, i.e., the initial state of the problem to be solved. Nodes are branched during the search process, generating children nodes more constrained than their father node. As shown in Figure 1, the generated nodes are evaluated, and then, the valid and feasible ones are stored in a data structure called *Active Set*.

At each iteration, a node is removed from the active set according to a selection rule [1]. The search generates and evaluates nodes until the data structure is empty or another termination criterion is satisfied. If an undesirable state is reached, the algorithm discards this node and then chooses an unexplored

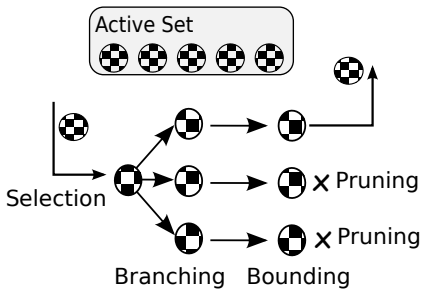


Figure 1: Visual representation of a B&B search algorithm (Own representation based on [3]).

(frontier) node in the active set. This action prunes some regions of the solution space, preventing the algorithm from unnecessary computation. The degree of parallelism of tree-based search algorithms is potentially very high, as the solution space can be partitioned into a large number of disjoint portions, which can be explored in parallel.

2.2.1. Branch-and-Bound Search Algorithms

Branch-and-Bound tree search algorithms are one of the most widely employed methods for solving combinatorial optimization problems to optimality. At each iteration, a B&B algorithm uses four basic operators (branching, bounding, pruning, and selection) to explore a usually huge search space intelligently. The best solution found so far is saved and can be improved from an iteration to another. The four operators of a B&B algorithm are described as follows.

- The **branching** operator divides a subproblem into several smaller, pairwise disjoint subproblems.
- The **bounding** operator is used to compute a lower-bound value of the optimal solution of each generated subproblem.
- The **pruning** operator uses the lower bound to decide whether to eliminate a subproblem or to continue its exploration. A subproblem s is eliminated (pruned) if $LB(s) \geq f(\pi^*)$, where LB designates the lower bounding function, f the objective function to minimize and π^* the best solution found so far (incumbent).
- The **selection** operator chooses one subproblem among all pending subproblems according to a predefined exploration strategy. In this paper, depth-first search (DFS) is used (backtracking), as memory requirements for other search strategies like best- or breadth-first search are often excessive [1].

In most B&B algorithms, the bounding operator is by far the most time-consuming part. If multiple lower bounds are available for a given problem,

one has to consider the trade-off between the computational complexity of a bound and the size of the explored search tree. Stronger (and computationally more expensive) lower bounds result in a more coarse-grained workload, while weaker bounds lead to larger trees composed of easy-to-evaluate subproblems. The choice of the bounding functions also has a strong impact on the irregular and unpredictable shape of the explored tree - thus, on load imbalance. Indeed, the performance of parallel B&B strongly depends on the efficiency of the load balancing strategy.

2.3. Branch-and-Bound Applied to the FSP

This section brings background information concerning the B&B applied to the Flow-shop scheduling problem (FSP). Initially, the FSP is introduced, followed by the bounding functions used for node evaluation. Next, two implementation aspects of a B&B algorithm for solving the FSP are detailed: data structures and the search procedure.

2.3.1. Problem overview

This work focuses on permutation combinatorial optimization problems, for which an N-sized permutation represents a valid and complete solution. Permutation combinatorial problems are used to model diverse real-world situations, and they are often NP-hard [26, 1]. The FSP consists in scheduling N jobs on M machines m_1, m_2,..., m_M in that order. The machines can handle at most one job at a time, and the processing of jobs cannot be interrupted. The objective is to minimize the makespan, *i.e.*, the termination date of the last job on the last machine. Although optimal non-permutation schedules exist, the search space is commonly restricted to permutation schedules. Even with this simplification, FSP is NP-hard for $M \geq 3$ [27].

2.3.2. The Bounding Function

Considering FSP as a test-case, we will now briefly describe the two lower bounds on the optimal makespan of a subproblem used in this work. A more detailed description can be found in the framework of lower bounds for the FSP

proposed in [28]. Let $p_{j,k}$ designate the processing time for job j on machine k and σ a subproblem defined by a partial permutation of jobs scheduled in the beginning. After the completion of the initial schedule, each machine will remain active at least for a time equal to the total remaining workload on that machine. Consequently, a lower bound is given by

$$LB1(\sigma) = \max_k \left(C(\sigma, k) + \sum_{j \in J} p_{j,k} + \gamma_k \right)$$

where $C(\sigma, k)$ designates the completion time of the partial schedule σ on machine k, J the set of unscheduled jobs and γ_k a precomputed term that corresponds to the minimum time required between the end of operations on machine k and the termination on the last machine (M). The computational complexity of the lower bound LB1 is $\mathcal{O}(M \times N)$.

A stronger lower bound, which we denote LB2, is obtained by relaxing capacity constraints on all but two bottleneck machines and taking the maximum makespan of the resulting $\frac{M(M-1)}{2}$ two-machine problems. Sorting jobs according to Johnson's rule [29] for all machine pairs is performed as a preliminary task, which reduces the complexity of the lower bound LB2 from $\mathcal{O}(M^2 N \log N)$ to $\mathcal{O}(M^2 N)$.

We decided to carry out experiments with two different lower bounds in order to gain insights into the impact of their different computational characteristics. More precisely, the simple lower bound LB1 results in larger trees with a more fine-grained workload. The two-machine bound LB2 results in a more coarse-grained but also more irregular workload, due to the improved efficiency of the pruning operator. Both lower bounds are implemented in C.

2.3.3. Data Structures and Search Procedure

The data structure `Node` is similar to any permutation combinatorial problem. It contains two integer vectors of the size of the complete solution. In the scope of the FSP, both vectors are of size $I.jobs$. The first vector, called *permutation*, keeps a feasible and valid incomplete solution, while the second

Algorithm 1: Depth-first B&B algorithm [2].

```
1  I ← get_instance()
2  upper_bound ← get_initial_solution(I)
3  stack.push(get_initial_node(I))
4  repeat
5  |  node ← stack.pop()
6  |  node.cost ← lower_bound(node, I)
7  |  if node is not a complete solution then
8  |  |  if node.cost < upper_bound then
9  |  |  |  generate the children nodes of node
10 |  |  |  push the generated children nodes onto the stack
11 |  |  end
12 |  else
13 |  |  if node.cost < upper_bound then
14 |  |  |  upper_bound ← node.cost
15 |  |  end
16 |  end
17 until stack is empty
```

one, called *scheduled*, keeps track of the already scheduled jobs by setting its position j to 1 if job j is scheduled. In turn, the instance I contains a matrix $C_{N\times M}$ of integers that gives the processing time $p_{j,k}$ of a job j ($j = 1, \ldots, N$) on machine k ($k = 1, \ldots, M$).

The search procedure is based on a serial and hand-optimized backtracking for solving permutation combinatorial problems originally written in C [13]. The serial backtracking was then adapted to Chapel, obeying the hand-made optimizations, instruction-level parallelism, data structures, and types. The search strategy is a non-recursive backtracking that does not use dynamic data structures, such as stacks. The semantics of a stack is obtained by using a variable *depth* and by trying to increment the value of the vector *permutation* at position *depth*. If this increment results in a valid incomplete solution, its feasibility is checked. In case the current incomplete solution is also valid, the *depth* variable is incremented, and the search proceeds to the next depth. After all configurations for a given depth are explored, the search backtracks to the previous one. One can see in Algorithm 1 a high-level design of the search strategy implemented in this work.

2.4. Related Works

Due to the highly irregular nature and prohibitive execution times of B&B algorithms for COPs, their parallelization has received these two last decades great attention from both combinatorial optimization and parallel computing communities. Indeed, big efforts have been made to revisit the parallelization of B&B following the rapid evolution of high-performance computing technologies[1] and their associated scientific and technical challenges. Indeed, B&B has been revisited for computational grids [4, 8] in the late 1990s and early 2000s, for multi-core processors [5] in the mid-2000s for many-core processors including GPU accelerators [6] and Intel Xeon Phi coprocessors and their combination [7], etc. However, these parallelization efforts have always been guided by the performance objective setting aside programming productivity. Indeed, the focus is mostly put on the design of new, often problem-specific, data structures [11, 12, 13, 5] for the efficient management of the "tsunami" of sub-problems generated during the resolution process and the proposition of new optimization techniques to deal with challenging issues including dynamic load balancing, communication optimization, synchronization, etc. These parallelization efforts are often limited to one or two specific hardware resource(s), which is obviously not sufficient to harness modern supercomputers. Indeed, these latter are increasingly large and include more and more heterogeneous devices making their programming more complex. Therefore, in addition to performance, productivity is a major criterion that should be considered when designing ultra-scale parallel applications like B&B.

Targeting performance, parallel tree-based search algorithms including B&B are very often written in C/C++, due to their low-level features and supported parallel computing libraries [30]. In a distributed context, these algorithms are combined with distributed programming libraries for the implementation of load balancing and the explicit communication between processes [3, 31, 4]. As a

[1] as attested by the Top500 international ranking of the 500 most powerful supercomputers in the World.

consequence, programming distributed tree search algorithms can be challenging and time-demanding. Therefore, high-level PGAS-based programming environments are good candidates for improving productivity. However, for B&B (exact optimization) the PGAS-oriented approach has never been investigated, which makes our contribution pioneering to the best of our knowledge. More generally, in the parallel optimization setting the rare papers we have found are [32, 33], which are related to parallel local search (PLS) metaheuristics (approximate optimization). In [32], the authors investigate the use of Global Address Space Programming Interface (GPI) PGAS API [34] for PLS. According to the reported experimental results, GPI allows one to get speed-ups similar to those obtained using MPI. In [33], the X10 [35] general-purpose language, developed by IBM, is used for PLS. X10 is based on Asynchronous PGAS and supports different levels of concurrency. The reported results show that good speed-ups could be obtained on some basic problem instances. However, no comparison to MPI(+X) is given as it is out of the scope of the paper.

3. A Productivity-aware Branch-and-Bound Algorithm

This section presents a distributed B&B algorithm for solving instances of the FSP to optimality that applies the productivity-aware features of Chapel. Initially, the initial premises considered in the design and implementation of the algorithm are discussed. Then, the main steps of the algorithm are detailed: the initial serial search and the distributed B&B.

3.1. Initial Premises

The main challenge in the conception of a B&B algorithm using a high-productivity language is to find a trade-off between productivity and performance. However, as stated in the last section, programmers have sacrificed productivity to achieve performance and to cope with the massive number of subproblems generated during the search. To achieve such a trade-off, we proceed as follows.

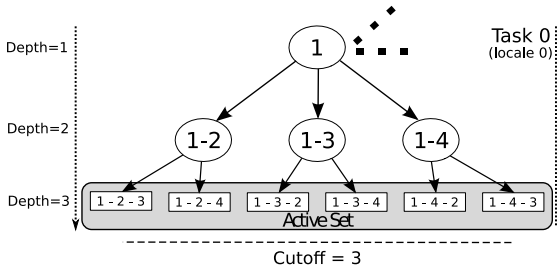


Figure 2: Initial search that generates the active set A for $jobs = 4$ and $cutoff = 3$. The figure depicts the branch that has the element 1 of the permutation as the root and generated 6 valid and feasible incomplete solutions at depth $cutoff = 3$.

The bounding operator of a B&B algorithm is problem-dependent and often a complex algorithm by itself. Therefore, translating a bounding function from C/C++ to Chapel would be time-consuming and error-prone. In this work, we exploit the C-interoperability features of Chapel. The bounding function used is a legacy code implemented in C, whereas all the search elements, i.e., the enumerative aspects of the search and the coherency of the incumbent solution, are implemented in Chapel. Furthermore, the built-in load balancing features of Chapel are used on both inter- and intra-node levels of parallelism.

The proposed algorithm consists of two main parts: the initial serial search on *locale* 0 and the multi-locale (distributed) search. The pseudo-codes presented in the following sections contain elements related to the Chapel language. Finally, the concepts further presented are similar to any permutation combinatorial problem and can be adapted for solving other problems with straightforward modifications [6, 13].

3.2. The Initial Search

Thanks to Chapel's global view of the control flow, it is not necessary to implement the SPMD programming model. Instead, the search starts serially, with *task* 0 running on *locale* 0. As one can see in Algorithm 2, *task* 0 initially receives an instance I of the problem, the first cutoff depth, and the second cutoff depth (*lines* $1-3$). As illustrated in Figure 2, a cutoff of c means that the initial search enumerates all feasible and valid incomplete solutions containing c

Algorithm 2: Initial Search on *task* 0, *locale* 0.

1 $I \leftarrow get\_instance()$
2 $cutoff \leftarrow get\_cutoff\_depth()$
3 $second\_cutoff \leftarrow get\_scnd\_cutoff\_depth()$
4 $ub \leftarrow calculate\_initial\_solution(I)$
5 $N \leftarrow I.jobs$
6 $metrics \leftarrow (0, 0)$
7 $max_{cutoff} \leftarrow \frac{N!}{(N-cutoff)!}$
8 $A \leftarrow [max_{cutoff}]\ Node$
9 $metrics += generate\_initial\_active\_set(cutoff, I, ub, A)$

elements of the permutation. Then, to make the pruning process more efficient, an initial upper bound (complete and valid solution) for instance, I is received or calculated (*lines* 4).

Before the initial search, it is required to define an active set A with its size equal to the maximum possible number of feasible and valid incomplete solutions at depth $cutoff$ (max_{cutoff}) (*line* 8). Next, the initial B&B generates a sufficiently large workload for the distributed search (*line* 9). For this purpose, *task* 0 searches from depth 1 (initial problem configuration) until the cutoff depth $cutoff$, storing all feasible and valid incomplete solutions at depth $cutoff$ into the active set A (*line* 9). The second cutoff depth will be used further.

3.3. Data Replication

Due to the global view programming model, the variables of Algorithm 2 are visible to tasks on other locales. However, to avoid remote reads and issues concerning the lower bound implementation written in C, read-only data needs to be replicated and initialized on each locale $i \in \{0, ..., L-1\}$, where L is the number of locales on which the application is going to run.

The pseudo-code in Algorithm 3 shows the initialization and replication of data structures required by the multi-locale search. Initially, a set of instances, of atomic upper bounds, and read-only data are defined using the *Private distribution*, which maps each index $i \in \{0, ..., L-1\}$ to `Locales[i]`. Then, three

Algorithm 3: Data replication algorithm.

```
1 PrivateSpace ← domain(1) mapped according to Private()
2 Instances ← [PrivateSpace] I_type;
3 read_only ← [PrivateSpace] RoData;
4 Upper_bounds ← [PrivateSpace] atomic int
5 forall inst in Instances, r in Read_only, and upper in Upper_bounds do
6 |   inst ← I
7 |   r ← start_read_only(Instances[here.id])
8 |   upper.write(ub)
9 end
```

loops take place for initializing these sets. The first one initializes each instance in the set *Instances* using the values of the one located on locale zero, whereas the second loop initializes the ready-only data requested by the bounding function on each locale. Finally, the third one is responsible for initializing an atomic upper bound one each locale. As all sets are defined using the *Private* distribution, each iteration i of the loop (*line* 5) is executed on locale i.

It is worth to mention that the built-in variable `here` present in (*line* 7) is of type `locale`. The `.id` query returns the identification $id \in \{0, ..., L-1\}$ of the locale on which the loop iteration is being executed. Finally, as *upper* is an atomic variable, it can only be written through the `write()` method (*line* 8) .

3.4. Distributing the Active Set

For distributing the active set A across several locales, it is required to define a domain and map it onto locales according to a distribution. As stated in Section 2.1, a distribution indicates how indexes of a data structure are mapped onto locales [20]. One can see in Algorithm 4 the steps required for the initialization and distribution of the PGAS-based active set.

Initially, the domain *Size* is defined using as a parameter the number of feasible and valid incomplete solutions in the task-local active set A (*line 1*). Then, *Size* is mapped according to a *standard distribution* (*line 2*) [2]. Next,

[2] https://chapel-lang.org/docs/modules/layoutdist.html

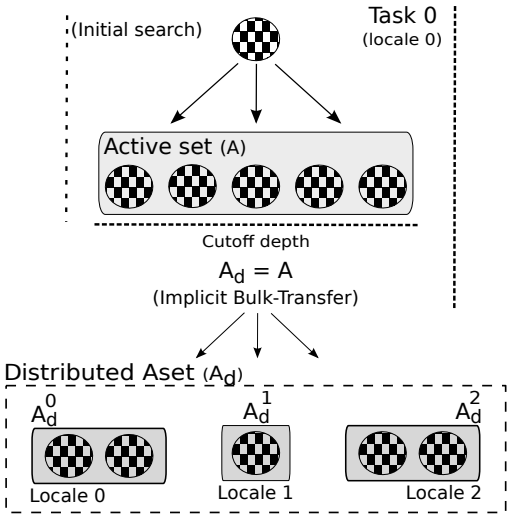


Figure 3: Task 0 is responsible for distributing the active set across several locales. The distributed active set A_d consists of several sets $A_d^i, i \in \{0, ..., L-1\}$, where L is the number of locales on which the application is going to run.

Algorithm 4: Active set distribution.

```
1 Size ← {0..(|A| − 1)} // Domain
2 D ← Size mapped onto locales according to a standard distribution
3 A_d ← [D] : Node
4 A_d ← A // Using implicit bulk-transfer
```

the distributed active set A_d of type `Node` is defined over D (*line 3*), and it receives the nodes of the task-local active set A though an implicit bulk-transfer (*line 4*). This way, instead of performing $|A|$ small transfers, only $L-1$ bulk transfers are performed through the network, being L the number of locales on which the application is going to run. Finally, as shown in Figure 3, the distributed active set A_d is an abstraction consisting in the union of several sets $A_d^i, i \in \{0, ..., L-1\}$.

3.5. The Multi-locale Search

After distributing the active set across different locales, the multi-locale search takes place. As one can see in Algorithm 5, parallelism is added in a way similar to OpenMP shared memory programming, through a `forall` clause,

applying a *master-worker* model. Moreover, also similarly to OpenMP, there is no need for implementing distributed load balancing, which is performed by using distributed iterators (`DistributedIters`).

In the master-worker model, *task* 0 (*locale* 0) is responsible for distributing *chunks* of nodes to other locales. Moreover, the searches are independent, and the nodes received are roots of B&B trees. Metrics are reduced through *Reduce Intents* (`with` keyword, *line* 1). In Chapel, it is possible to use the `Tuple` data type (equivalent to C-structs) and reduce all metrics at once (*line* 2). Differently from OpenMP, it is not required to define a tuple reduction. The distributed search finishes when the active set A_d is empty. Then, the program presents a report taking into account the metrics collected and the found optimal solution.

3.5.1. Exploiting Intra-locale Parallelism

It is possible to perform a multi-locale search relying only on the `forall` loop of Algorithm 5 (*line* 1). The generated code exploits two levels of parallelism: inter-locale (distributed) and intra-locale (the CPU cores a locale has). In such situation, a task receives a chunk of nodes from the master, and then the B&B search proceeds from depth $cutoff$ until the depth N, *i.e.* the number of jobs. However, the number of feasible and valid incomplete solutions at depth $cutoff$ may be insufficient to efficiently use all CPU cores of several locales at once. To cope with this situation, we proceed as outlined in Algorithm 6.

Initially, we calculate the maximum number of children nodes the node received as a parameter can have at depth $second\_cutoff$, further referred to by $max\_children$ (*line 6*). In Algorithm 6, a *child node* is a valid and feasible incomplete solution at depth *second_depth* which has *node* as its ancestor

Algorithm 5: Launching the multi-locale search.

```
1 forall node in A_d following a distributed iterator with(+ reduce metrics) do
2 |   metrics += distributed_BB(node, cutoff, second_cutoff,
3 |       Instances[here.id], Upper_bound[here.id]);
4 end
5 show_results(metrics, Upper_bound)
```

Algorithm 6: Exploiting intra-node parallelism.

Input: $node$, $cutoff$, $second\_cutoff$, I^i, and $upper\_bound^i$

Output: A *tuple* containing the explored tree size and the number of solutions found.

```
1  N ← I^i.jobs
2  metrics ← (0, 0)
3  upper ← upper_bound^i.read()
4  max_first ← (N)! / (N − cutoff)!
5  max_second ← (N)! / (N − second_cutoff)!
6  max_children ← max_second / max_first
7  A_t ← [max_children] Node // Task-local active set.
8  metrics += generate_task_local_active_set(node,
9      cutoff, second_cutoff, I^i, upper, A_t);
10 forall n in A_t following an iterator with(+ reduce metrics) do
11 |  metrics += local_BB(n, I^i, second_cutoff, upper_bound^i);
12 end
13 return metrics
```

at depth *cutoff*. Before the intra-locale B&B, a task-local active set A_t of size $max\_children$ is defined. Next, a second search is performed from depth $cutoff$ until $second\_cutoff$, also storing into A_t all children nodes (*line* 8). Finally, the task-local B&B takes place (*lines* $10-12$), searching from depth $second\_cutoff$ until N (number of jobs), also applying the master-worker model of Algorithm 5. In this case, regular iterators are used instead of the distributed ones. The task-local B&B also finishes when A_t is empty. The strategy of performing several partial searches using two or more cutoff depths is similar to load balancing strategies applied in the context of GPU-accelerated tree search [13].

3.6. Global-view Consistency of the Incumbent Solution

Thanks to Chapel's global view of the control flow, it is possible to access an atomic variable in the same way it is accessed in a shared memory environment, regardless the locale on which it is located. This should be interesting in

Algorithm 7: Overview of the multi-locale B&B algorithm.

```
1  (I, cutoff, second_cutoff) ← get_problem( )
2  ub ← calculate_initial_solution(I)
3  A ← generate_initial_active_set(cutoff, I, ub)
4  (Instances, Upper_bound) ← replicate_readOnly_data(I, ub)
5  A^d ← distribute_active_set(A)
6  forall node in A_d following a distributed iterator with(+ reduce metrics) do
7  |   distributed_BB(node, cutoff, second_cutoff,
8  |       Instances[here.id], Upper_bound[here.id]);
9  end
10 show_results(metrics, Upper_bound)
```

case one has at his/her disposal a Cray system that supports network-atomics. Otherwise, accessing atomic variables placed on a different locale works like a remote procedure call, being executed on the locale on which the atomic variable is declared.

To avoid such an overhead, tasks on *locale i* search using the $Upper\_bounds[i]$ atomic variable. Moreover, to ensure that all locales can access the best solution found so far, and to return an optimal solution for the instance at hand, an incumbent atomic solution is also declared on locale 0. When the search called in Algorithm 6 (*line* 11) finds a new solution, it attempts to update both global and local incumbent solutions through the *minExchange* operation. If it is not possible to update neither the former nor the latter, the *minExchange* function returns to the task the smallest value globally found so far.

3.7. Overview of the Proposed Algorithm

Algorithm 7 is an overview of the proposed distributed B&B, and its lines correspond to high-level representations of the algorithms previously detailed. Algorithm 2 corresponds to lines $1-3$. In turn, the read-only data replication and the active set distribution correspond to lines 4 and 5, respectively. Finally, lines 6 to 10 correspond to Algorithm 5. The intra-locale parallelism and the coherency of the incumbent solution take place inside the subroutines of Algorithm 6.

4. Performance Evaluation

This section presents a performance-oriented evaluation of the B&B algorithm previously proposed, and it is organized as follows. Section 4.1 defines the experimental protocol and Section 4.2 brings the parameter settings. The performance and scalability results are presented and analyzed in Section 4.3 and Section 4.4, respectively.

4.1. Experimental Protocol

In this evaluation, the following B&B implementations are considered:

- **Chapel-BB**: implementation of the productivity-aware distributed B&B presented in the last section. As previously mentioned, the used bounding function is a legacy code implemented in C, whereas all other features of the search are implemented in Chapel.

- **MPI-PBB**: single program - multiple data (SPMD) B&B written in C++ and MPI+Pthreads. MPI-PBB is a Master-Worker algorithm using an interval-based encoding of work units and the IVM data structure [6] for the implementation of depth-first search. Each worker consists of multiple worker threads performing local work stealing operations for load balancing on the intra-node level. A dedicated thread is used for communication with the master process, allowing to overlap work progress and communication efficiently. Further details can be found in [36] and some implementation aspects are discussed in Section 6.1.

It is worth to mention that both implementations follow the *master-worker* parallel model, use the DFS search strategy as selection rule, and implement the bounding functions introduced in Section 2.3.2. However, *Chapel-BB* is not a translation of *MPI-PBB*. The data structures and the used bounding functions differ between them. For a fair comparison, both implementations enumerate equivalent search spaces for proving the optimality of a solution.

In this section, we investigate the single and multi-locale performance of Chapel-BB, its scalability according to the number of locales, as well as the influence of the PGAS data structure distribution on the execution time. We also study the impact of the built-in distributed load balancing strategies on the overall performance of the application. Due to the massive amount of collected data, some results are presented in a summarized way.

4.2. Parameters Settings

The benchmark instances used in our experiments are the FSP instances defined by Taillard [37]. We use only the 10 instances where $M = N = 20$. For most instances where $M = 5$ or 10, the bounding operator gives such good LBs that it is possible to solve them in few seconds using a sequential B&B. Instances with $M = 20$ and $N = 50, 100, 200$ or 500 are very hard to solve. For example, the resolution of the instance *Ta056* ($N = 50$, $M = 20$), performed in [4], lasted 25 days with an average of 328 processors and a cumulative computation time of about 22 years.

To compare the performance of two B&B algorithms, both should explore the same search space. When an instance is solved twice using a parallel tree search algorithm, the number of explored nodes varies between two resolutions. Therefore, for all instances, the initial upper bound (cost of the best found solution) is set to the optimal value, and the search proves the optimality of this solution. This initialization ensures that precisely the critical sub-tree is explored, *i.e.*, the nodes visited are exactly those nodes which have a lower bound smaller than the optimal solution [38].

All computer nodes are symmetric and operate under Debian 4.9.0, 64 bits. They are equipped with *two* Intel Xeon Gold 6130 @ 2.10 GHz (a total of 32 cores/64 threads per node) and 192 *GB* RAM. Thus, up to 1024 cores/2048 threads are used in the experiments. All locales are interconnected through a 100 *Gbps* Intel Omni-Path network. The number of locales in the experiments ranges from 1 to 32, and the application is the same for either one or more than one computer node(s). The number of locales is passed to the B&B using

Table 1: Summary of the environment configuration for multi-locale execution and compilation.

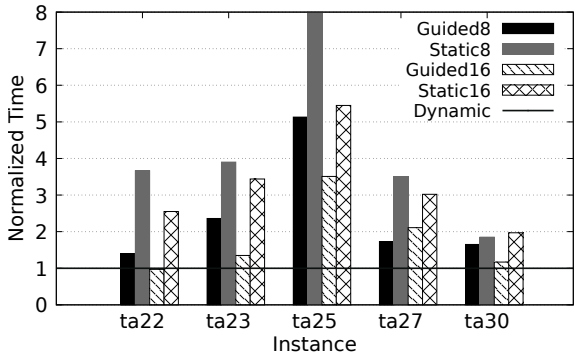
Variable	Value
CHPL_RT_NUM_THREADS_PER_LOCALE	64
CHPL_TARGET_ARCH	*native*
CHPL_COMM	*gasnet*
CHPL_GASNET_SEGMENT	*everything*
CHPL_COMM_SUBSTRATE	*ofi*
GASNET_PSM_SPAWNER	*ssh*

Chapel's built-in command line parameter `-nl L`, where L is the number of locales on which the application is executed.

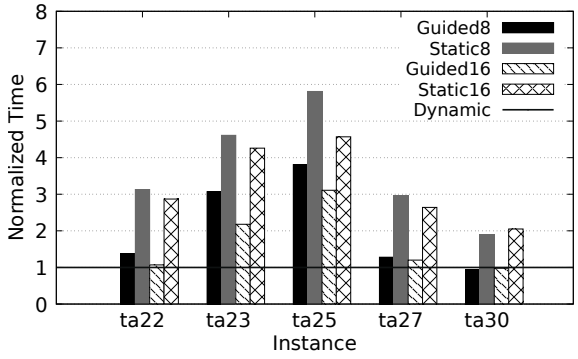
In MPI-PBB, each worker can be configured to use any number of worker threads. We chose to use 8 threads (plus one additional communication thread) per MPI process. Therefore, a total of $8L$ processes is launched and mapped evenly across the L compute nodes using the `-map-by:8:node` option for `mpirun`. Node 0 runs the master process using the same configuration, meaning that node 0 hosts at most 7 worker processes (56 threads).

The Chapel implementation was programmed for version 1.19, and the *default* task layer (qthreads) is the one employed. Chapel's multi-locale code runs on top of GASNet, and several environment variables should be set with the characteristics of the system the multi-locale code is supposed to run. One can see in Table 1 a summary of the runtime configurations for multi-locale execution. The OpenFabrics GASNet implementation is the one used for communication (`CHPL_COMM_SUBSTRATE`) along with SSH, which is responsible for getting the executables running on different locales (`GASNET_PSM_SPAWNER`). Concerning MPI+PBB, OpenMPI 2.0.2 with `MPI_THREAD_MULTIPLE` support and along with *gcc* 6.3.0 were used for compilation and execution.

Chapel provides several standard distributions to map data structures onto locales. Different tests were also carried out to identify the best option in the context of this work. The one chosen was the one-dimension *BlockDist*, which horizontally maps elements across locales. For instance, in case $L = 3$ and $|A_d| = 8$, elements $0, ..., 2$ are on locale l_0, $3, .., 5$ on locale l_1, and $6, 7$ on locale l_2. In the scope of the present research, choosing a different standard distribution does not lead to performance improvements.



(a) LB1



(b) LB2

Figure 4: Comparison of all three distributed load balancing schemes for (a) LB1 and (b) LB2 executed on 8 and 16 locales (256 and 512 cores, respectively).

Experiments were carried out to choose suitable cutoff depths (Algorithm 2, *lines* 2 *and* 3). The first cutoff depth directly influences the size of A_d, and therefore the time spent in distributing the active set across locales. Moreover, this parameter also influences the granularity of the search. The choice of such a parameter is difficult because each instance has its own characteristics, and therefore, the size of A_d at depth $cutoff$ varies. According to preliminary experiments, the best overall results are observed for $cutoff = 4$ and $second\_cutoff = 7$. Table 2 summarizes the best parameters experimentally found for Chapel-BB.

Chapel provides two different distributed load balancing iterators: *guided*

Table 2: The Best parameters experimentally found for the B&B implementation written in Chapel.

Parameter	LB1	LB2
$cutoff$	4	4
$second\_cutoff$	7	7
Distributed chunk	32	8
Intra-node chunk	4	2

and *dynamic*, which are also similar to OpenMP's schedules of the same name. Experiments were also carried out to identify the best *chunk* for both load balancing strategies. As depicted in Figure 4, using the dynamic distributed iterator results in the best overall performance. Therefore, it is the iterator to be considered hereafter in the results presented for Chapel-BB.

As mentioned in Section 3, it is an option to exploit two levels of parallelism by just executing the *forall* loop of Algorithm 5 (*line 1*). This version of the implementation is further referred to as *Regular*. However, as one can see in Figure 5, relying on the code automatically generated by the compiler for exploiting two levels of parallelism results in inefficient use of the computational resources. Implementing the intra-node parallelism results in a distributed B&B from 2× to 5× faster than its counterpart that relies on the compiler (Regular implementation).

Finally, due to the global view of the control flow, it is possible to access

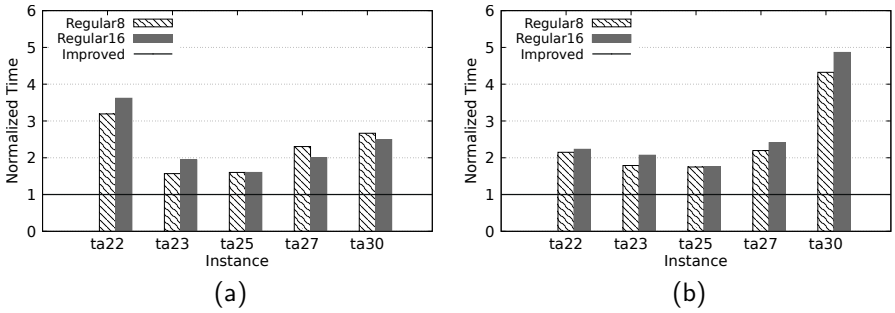


Figure 5: Comparison between the *Regular* Chapel-BB version and the *Improved* one, which has its intra-node parallelism programmed by hand. Results shown are for (a) LB1 and (b) LB2 executed on 8 (256 cores) and 16 locales (512 cores).

Table 3: Number of decomposed subproblems (NN) for proving the optimality of the initial solution received by both B&B implementations, using lower bounds LB1 and LB2 (in 10^6 nodes). Corresponding execution time (T) using 1 node (in seconds).

Instance-#	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**
$\mathbf{NN}_{LB1}$ (10^6)	711	37 200	71 876	5208	11 392	1854	12 285	3018	111
$\mathbf{T}_{LB1}$ (sec)	120	6400	11 460	970	1750	320	2100	490	20
$\mathbf{NN}_{LB2}$ (10^6)	14	114	2132	233	511	54	194	9	13
$\mathbf{T}_{LB2}$ (sec)	28	190	2930	395	680	112	292	18	19

an incumbent solution defined on locale 0 atomically. However, the atomic operations are performed on the locale on which the atomic variable was defined. According to preliminary experiments, the strategy of using a single atomic incumbent solution on locale 0 is unfeasible. For instance, it is 6× slower solving instance *ta*22 accessing a unique incumbent solution on locale 0 than using the strategy proposed in Section 3.6.

4.3. Performance Results

Table 3 reports the size number of decomposed subproblems for solving the chosen instances to the optimality with LB1 and LB2. The corresponding execution time on a single node is also shown for MPI-BB. Although the instances have the same number of machines and jobs, the search space enumerated to prove the optimality of the initial solution can vary considerably (*e.g.*, *ta*24 *vs.* *ta*30). Consequently, for MPI-PBB the execution times on a single node range from 18 seconds to more than 3 hours.

It is shown in Figure 7a the relative execution time of Chapel-BB compared to MPI-PBB for solving to the optimality the chosen instances on 1 to 32 locales. Taking into account LB1 and up to 2 computer nodes, Chapel-BB is from 8% to 22% faster than MPI-PBB, being only slower for the two smallest instances (*ta*22 and *ta*30) on 2 locales. The high single-node performance of Chapel is justified by the fact that it is a compiled language that allows one to program hand-optimized data structures when necessary. Moreover, the data structures used by Chapel-BB were designed for achieving performance in situations where the bounding function is not compute-intensive [39]. As LB1 is less compute-intensive than LB2, the fastest data structure of Chapel-BB pays off in such a

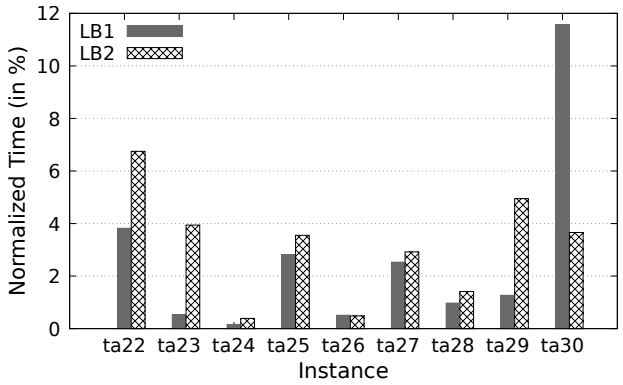


Figure 6: Proportion (in %) of the initialization and distribution of A_d compared to the whole execution time. Results are for lower bounds LB1 and LB2, executed on 32 locales

case. Finally, MPI-PBB reserves computational resources to be the coordinator of the search. Therefore, the smaller number of worker threads compared to Chapel-BB negatively impacts MPI-PBB's performance on up to 2 locales.

As the number of computer nodes increases, the load balancing becomes crucial for achieving performance. Taking into account 32 locales (1024 cores), Chapel-BB remains faster or at least equivalent to MPI-PBB for the three biggest instances ($ta23$, $ta24$ and $ta28$), and it is 21% and 31% slower for $ta25$ and $ta26$, respectively. As depicted in Figure 8a, the built-in load balancing provided by Chapel generates more regular loads among locales for these five instances. The ratio of the biggest load processed by a locale over the smallest one varies from 1.09× ($ta23$) to 1.72× ($ta26$). Additionally, as one can see in Figure 6, the distribution of the PGAS-based active set amounts for less than 2% of the execution time for the three biggest instances.

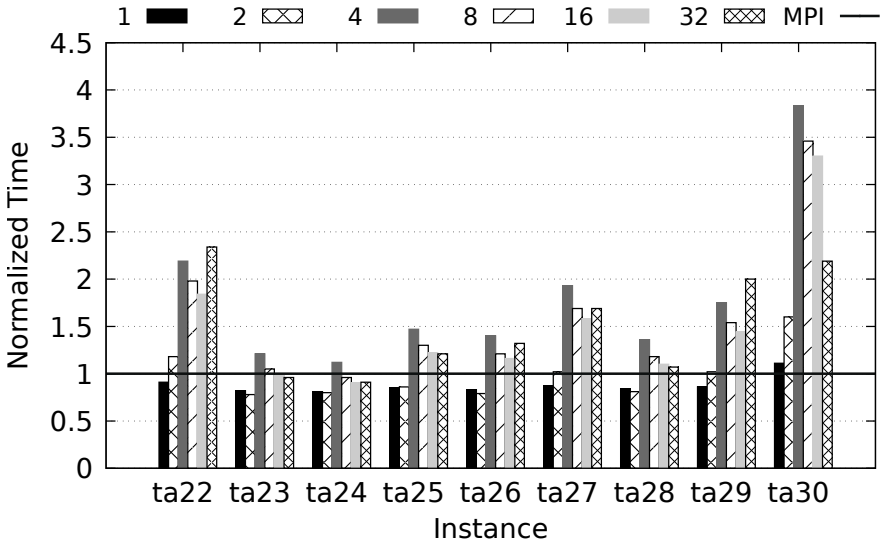
In turn, the built-in load balancing provided by Chapel cannot deliver to the 4 smallest instances ($ta22$, $ta27$, $ta29$ and $ta30$) regular loads among locales (Figure 8). Taking into account $ta22$ and $ta30$, the biggest load is 4.2× and 10× bigger than the smallest one, respectively. The distribution of the active set is also costly for these instances. It amounts for 4% ($ta22$) and 12% ($ta30$) of the execution time on 32 locales, as depicted in Figure 6. Consequently, Chapel-BB is from 1.68× ($ta27$) to 2.34× ($ta30$) slower than MPI-PBB on 32 locales. The

worst results are observed for $ta30$, which combines small search space with irregularity and costly active set distribution. However, these results were not seen for $ta30$ on 32 locales, because MPI-PBB also faces the same challenges as the number of locales increases.

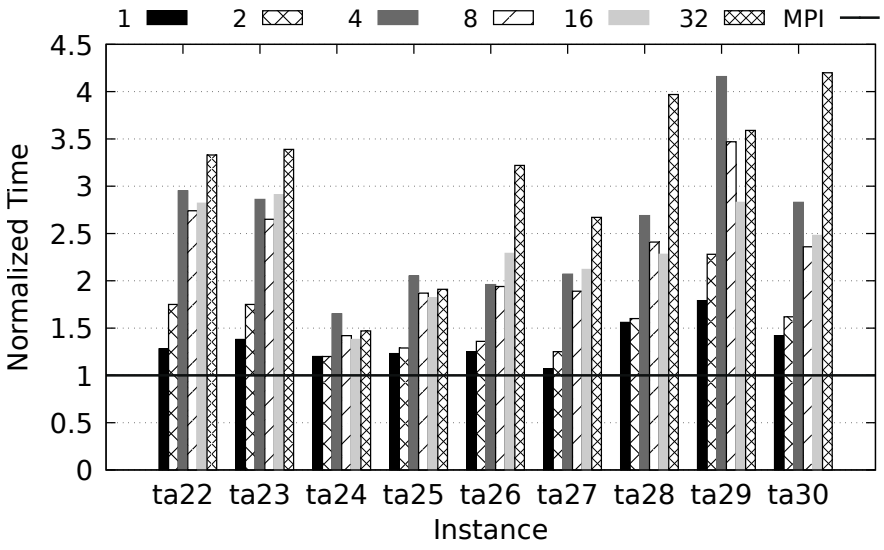
As one can see in Table 3, the 2-machine bound (LB2) is much stronger than LB1, resulting in smaller trees. This way, the distribution of A_d amounts more negatively for LB2 than for LB1 (refer to Figure 6). The behavior observed for LB1 when solving the smaller instances can be seen for all cases but $ta24$, which has the biggest solution space. Load imbalances are present even on 2 computer nodes. For instance, the value of the biggest over smallest load is 1.75× for $ta24$ (See Figure 8b). Moreover, as pointed out in Section 2.3.2, LB2 is much more costly to compute than LB1, which removes from Chapel-BB the benefits of a faster data structure. As a consequence, an equivalent execution time to MPI-PBB on one locale can be observed only for $ta27$. On two locales, Chapel-BB is from 1.20× ($ta24$) to 2.28× ($ta29$) slower than its counterpart, as depicted in Figure 7b. As the number of locales increases, the criticality of load imbalance increases as well. The value of the biggest load over the smallest one processed by a locale reaches 15.8× and 18.7× on 32 locales for $ta30$ and $ta22$, respectively. As depicted in Figure 7b, Chapel-BB is from 1.47× to 4.2× slower than MPI-PBB on 32 locales.

4.4. Scalability Analysis

As in Section 4.3, we first consider LB1. Figure 9a shows the speedups achieved by Chapel-BB and MPI-PBB on 2 to 32 locales. Results ranging from 19% ($ta30$) to 84.3% ($ta24$) of the linear speedup on 32 locales are observed for Chapel-BB. Speedups ranging from 60% ($ta25$) to 85% ($ta24$) are observed for those instances to which the built-in load balancing produces more regular loads, and the distribution of the active set is less costly. As for the execution time results and due to the same reasons, the higher speedups can be seen for $ta23$ and $ta24$. For the smaller instances ($ta22$, $ta27$, $ta29$ and $ta30$), severe speedup decreases are observed when comparing the results on 2 to the ones on 32 locales.

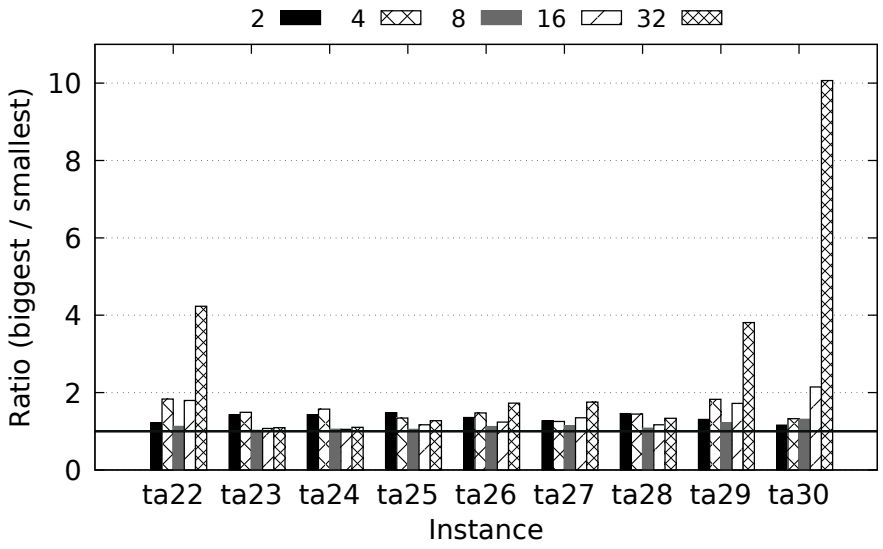


(a) LB1

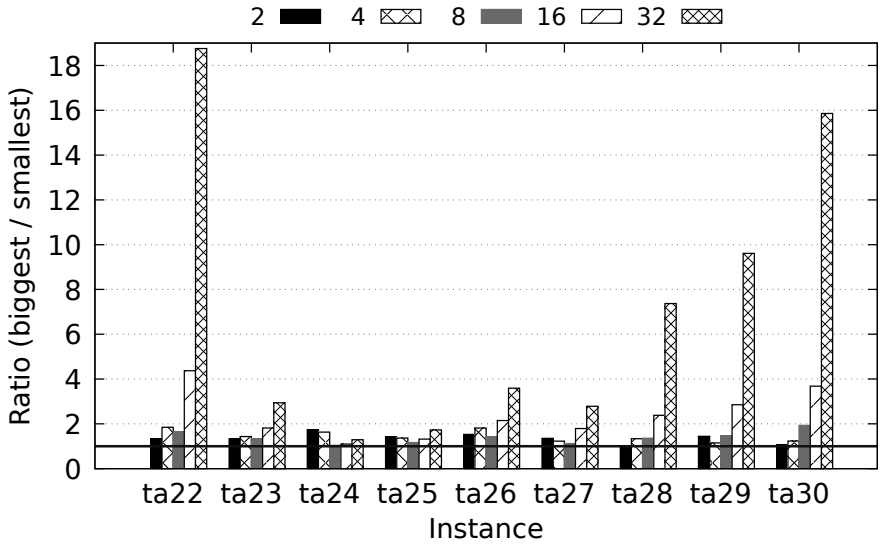


(b) LB2

Figure 7: Execution times of Chapel-BB for solving instances *ta*21-30 to optimality. Results shown are for (a) LB1 and (b) LB2 executed on 1 (32 cores) to 32 locales (1024 cores). For each $< instance, locale >$ configuration, the execution time is given relative to the MPI-PBB baseline.



(a) LB1



(b) LB2

Figure 8: Proportion of the biggest load over the smallest one produced by the *dynamic* distributed iterator provided by Chapel. In this case, the term *load* means the percentage of the solution space processed by a given locale. Results shown are for (a) LB1 and (b) LB2 executed on 2 (64 cores) to 32 locales (1024 cores).

For such subset of instances, results range from 30% ($ta30$) to 45% ($ta27$) of the linear speedup. In turn, the hand-programmed work stealing mechanism and communication of MPI-PBB pays off. The speedups achieved by the MPI-based search are greater than 80% of the linear one on 2 to 32 nodes. Severe speedup decreases are only observed for $ta22$ and $ta30$, due to the reasons already exposed in Section 4.3. Figure 10a shows the speedup reached by Chapel-BB compared to the results of MPI-PBB. The search written in Chapel reaches from 40% ($ta22$) to 92% ($ta24$) of the values observed for its MPI-based counterpart.

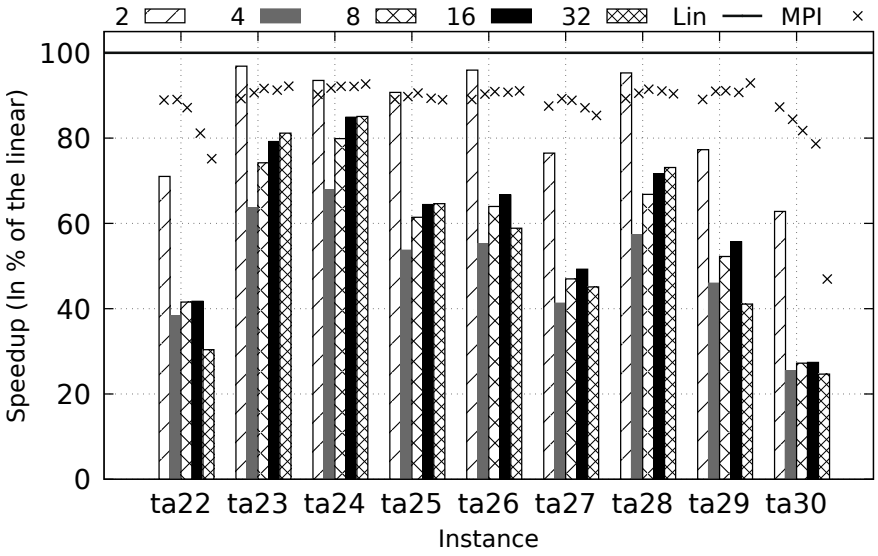
Taking into account LB2, results farther from the linear speedup are observed for both Chapel-BB and MPI-PBB. Moreover, when comparing to the results of LB1, severe speedup decreases are noted for the majority of instances. As can be seen in Figure 9b, MPI-PBB achieves from 38% to 93% of the linear speedup on 32 locales. However, severe speedup drops are observed for 55% of the instances. Considering $ta29$, the speedup decreases from 88% (2 locales) to 38.7% (32 locales), and the same behavior can be seen for $ta22$, $ta27$ and $ta30$. Therefore, this scenario with smaller trees and coarser granularity shows to be challenging even for the state of the art work stealing implemented by MPI-PBB. In turn, Chapel-BB achieves from 19% ($ta22$) to 78% ($ta24$) of the linear speedup on 32 locales, which represents from 35% to 85% of the values seen for MPI-PBB (refer to Figure 10b). Severe speedup drops are observed for 7 out of 9 instances.

5. Productivity-oriented Evaluation

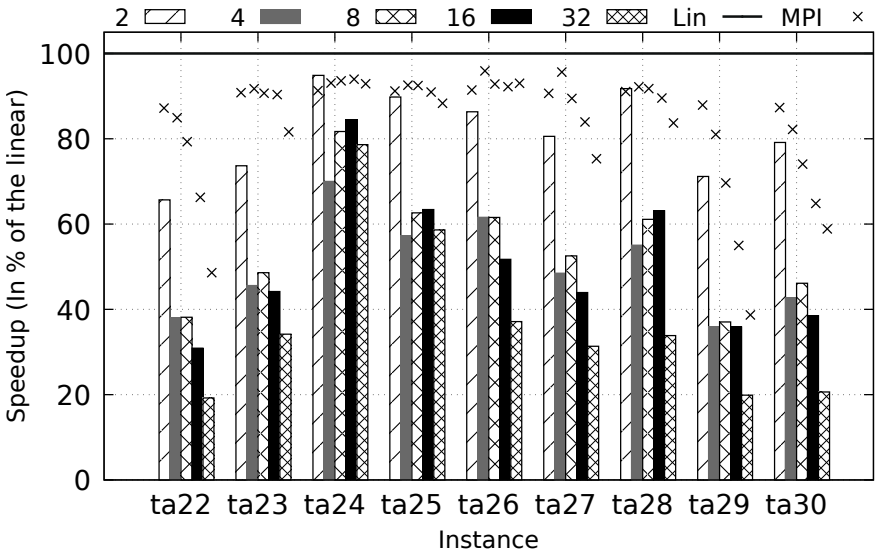
This section presents a productivity-oriented evaluation of Chapel for programming distributed B&B algorithms. Section 5.1 brings the experimental protocol and Section 5.2 brings the results.

5.1. Experimental Protocol

In this section, we use the model proposed by Snir and Bader for measuring productivity in high-performance computing [40]. Initially, consider *Utility* as

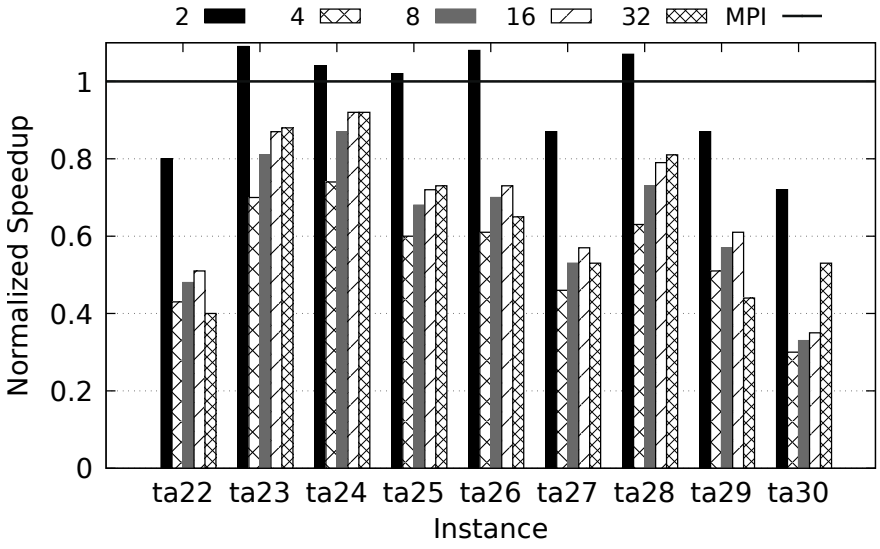


(a) LB1

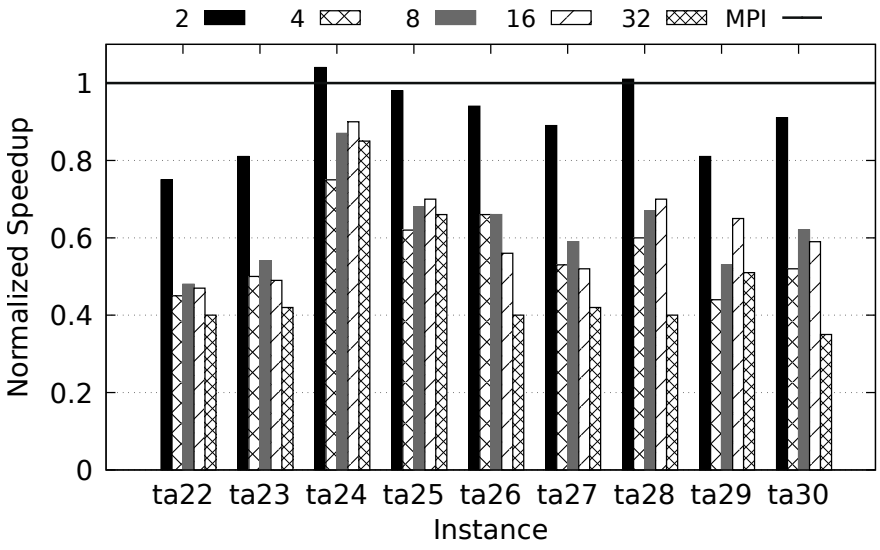


(b) LB2

Figure 9: Speedup achieved by Chapel-BB and MPI-PBB on 2 (64 cores) to 32 locales (1024 cores) compared to the execution on *one* locale. Values are given in percent of the linear speedup ($Lin - 100\%$). For a given $< instance,\ locale >$ configuration, the speedup achieved by MPI-PBB is presented through an ✗ mark. Results shown are for (a) LB1 and (b) LB2.



(a) LB1



(b) LB2

Figure 10: Speedup achieved by Chapel-BB on 2 (64 cores) to 32 locales (1024 cores) compared to the execution on *one* locale. Values are given based on the speedups achieved by MPI-PBB. Results are shown for (a) LB1 and (b) LB2.

the value received on getting an answer to a problem in a certain time [41]. According to the model, *Productivity* (ψ) is utility over a total cost, and it is defined as follows.

$$\psi = \frac{S_p \times E \times A}{C_s + C_o + C_M} \tag{1}$$

where:

- S_p : the *operations/time* peak that can be achieved on the system.
- E : efficiency achieved by the parallel program.
- A : availability of the system.
- C_s : software cost.
- C_M and C_o: cost of the machine and ownership, respectively. These metrics concern any cost related to energy, hardware maintenance, human resources, etc.

In this work, we consider both the machine and ownership costs as *zero*. These values are justified by the fact that the authors do not handle costs. Moreover, the availability value is 100%. The most challenging parameter to set is the software cost (C_s), which should reflect all costs involved in the design of the program.

The parameter C_s can be defined as:

$$C_s = c_m \times \Gamma \times r \tag{2}$$

where c_m is the monetary cost of the programmer, r is a measure of programming effort, such as time unit per line of code, and Γ is the size of the program. Hereafter, the size of the program is going to be measured in lines of codes (SLOC). In the context of the present work, it is difficult to measure the time required to produce each B&B implementation introduced in Section 4.1. Moreover, the time to program the Chapel-based B&B also includes the time needed

to learn several aspects of the language. Concerning the monetary costs, the languages used are freely available, and the programmers have a fixed cost. Therefore, for the sake of greater simplicity, the software cost is going to be hereafter considered as $C_s = \Gamma$. It is also difficult to calculate the SLOC in the context of this work. The two B&B implementations are programmed in different languages, and they follow distinct programming models. Additionally, the data structures differ between both applications. This way, we isolate the following code segments for calculating the software cost:

- Initialization of the distributed aspects of the search.
- Termination criteria checking.
- Operations involved in the coherency of the incumbent solution.
- Load balancing/work distribution.
- Second-level (intra-node) parallelism.

Once those code segments are isolated, non-essential parts such as comments, timers, includes, and print functions are removed. Moreover, as MPI-PBB is implemented in C++, the declarations inside the header files are not taken into account. As Chapel is a compiled language and declaration significantly amounts for SLOC, declarations of variables are also removed from Chapel-BB's SLOC count. One can see in Table 4 the SLOC count for Chapel-BB and MPI-PBB.

5.2. Productivity Results

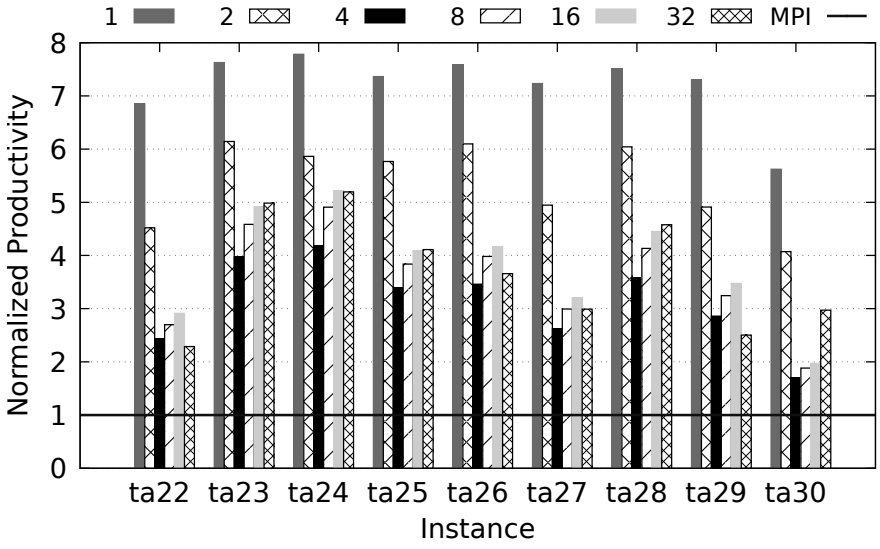
According to Table 4, the overall software cost of MPI-PBB is 5.6× higher than the one of Chapel-BB. The most expensive parts of the MPI-PBB code are the load balancing and termination criteria, which are 35× and 18× more costly than for Chapel-BB, respectively. Concerning the load balancing, the use of PGAS-based data structure hides several aspects of communication. Furthermore, it also allows the use of distributed iterators for load balancing in a way

Table 4: The SLOC count for Chapel-BB and MPI-BB, which represents the software cost parameter (C_s). The total SLOC is smaller than the sum of costs because some lines serve for more than one purpose.

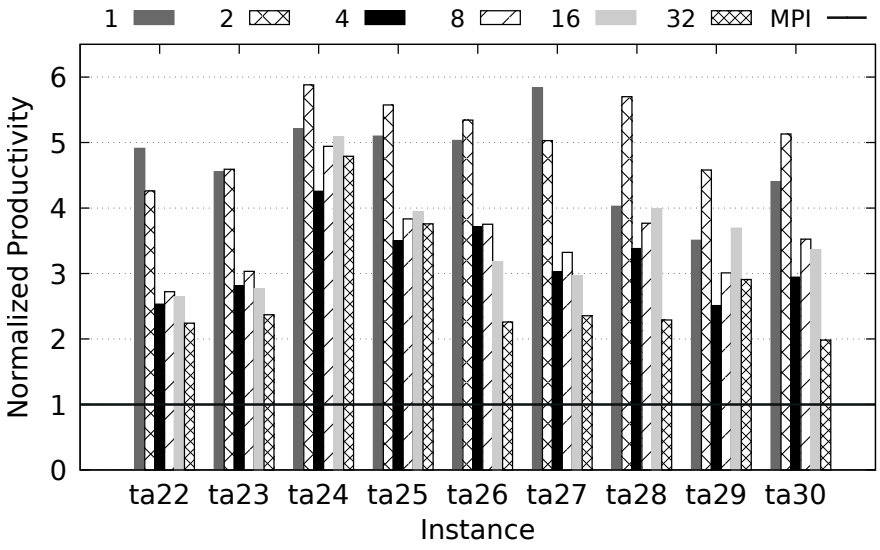
Segment of the code	Chapel-BB	MPI-PBB
Initialization	23	37
Incumbent solution	12	44
Metrics reduction	4	9
Load balancing	5	176
Second level of parallelism	12	72
Termination criteria	2	36
Total SLOC	53	300

similar to shared memory programming. Moreover, as shown in Algorithm 5, there is no need for programming termination criteria. The loop of *line* 1 finishes when the distributed active set is empty, and metrics are reduced by using a distributed reduction that is also hidden from the programmer. As shown in Table 4, the initialization of the search is the most costly part of Chapel-BB. It is responsible for the definition of the local active set, the initial search, the definition of the distribution, mapping A_d according to this distribution, and finally, it also distributes A_d across different locales. In turn, for MPI+Pthreads, the initialization takes into account several MPI routines and details concerning the IVM data structure and initialization of the workers and the master process.

The SLOC metric is not enough for measuring productivity, and it is related to the expressiveness power of Chapel compared to MPI+Pthreads [42]. Besides low software cost, a language must also allow the programmer to produce software that scales and is efficient to achieve high productivity [41]. One can see in Figure 11a the productivity (ψ) achieved by Chapel compared to MPI+Pthreads, taking into account LB1. As the results are given in terms of a utility value over the software cost, the highest results are observed for 1 and 2 locale(s). In such cases, Chapel's productivity is from 4.0× ($ta30$, 2 locales) to 7.7× ($ta24$, 2 locales) superior to the one achieved by MPI+Pthreads. As the number of locales increases, MPI-PBB presents better efficiency than Chapel-BB. Therefore, despite Chapel's low software cost, its productivity val-



(a) LB1



(b) LB2

Figure 11: Normalized productivity achieved by Chapel compared to MPI+Pthreads when implementing the B&B applications listed in the last section. Results shown are for (a) LB1 and (b) LB2, and given for each pair $< instance, locale >$ taking into account from 1 (32 cores) to 32 locales (1024 cores).

ues decrease. For instance, it is only 70% more productive than MPI+Pthreads for $ta30$ (4 locales), and the productivity observed for $ta22$ and $ta27$ drops for around 30% of the value reached on one locale. Taking into account LB1 and 32 locales, Chapel is from 2.2× to 5.2× more productive than MPI+Pthreads. Taking into account LB2 (Figure 11b), the highest productivity achieved by Chapel is also observed on up to 2 locales: from 3.5× ($ta29$, 1 locale) to 5.9× ($t24$, 2 locales). Then, the productivity values decrease as the node count increases. Moreover, as Chapel-BB is less efficient for LB2, the overall results for LB2 are also smaller than for LB1. Productivity values ranging from 2× ($ta30$) to 4.8× ($ta24$) are observed on 32 locales.

6. Discussion

6.1. Productivity

It is difficult to provide an exact number of SLOC indispensable for MPI-PBB. Considering only the core of the master process, MPI-based communication routines, and a basic multi-core worker using work stealing and a dedicated communication thread, the size of the code is between 1000 and 2000 lines. The fact that multiple synchronization points at the node level (work-stealing, termination of a workgroup, update of the incumbent) need to be handled asynchronously with two-sided MPI communications is challenging and error-prone as it induces several hard-to-detect race conditions. At the inter-node level work unit communications are interleaved with smaller, auxiliary messages for global termination detection, and sharing of the best solution found so far. This requires pre-receive queries of message types using `MPI_Probe` and separate send/receive routines for different message types. The same goes for the aggregation of metrics from worker threads, which requires rather significant programming efforts in MPI-PBB.

The total SLOC of Chapel-BB is much closer to the values of Table 4 than for MPI-PBB. Thanks to Chapel's global view of the control flow and data structures, the main difference between the multi- and single-locale versions lies in

the use of the PGAS data structures and distributed iterators. There is no need for explicitly dealing with communication between locales, termination criteria, and metrics reduction. Furthermore, there is no need for an additional library to exploit each level of parallelism. Moreover, the coherency of the incumbent solution is close to the way it is performed in shared-memory programming. These features greatly simplify the code and remove potential sources of errors.

The points discussed in the last paragraphs show that Chapel has a higher expressiveness power than MPI+Pthreads indeed, and also provides several features that might decrease the cost of programming a distributed B&B algorithm. However, the software cost used for calculating productivity does not take into account several time-consuming tasks involved in the design of Chapel-BB. For instance, the time spent on understanding the new programming models and learning the language. Moreover, in the context of this work, the program needs to be efficient to achieve high productivity. As MPI+X is a standard for HPC, the authors already know several best practices for achieving performance and scalability using MPI+X, which penalizes Chapel-related results. Chapel-BB could achieve better scalability and productivity in case the authors had more experience with the Chapel language.

6.2. Performance and Scalability

The performance results achieved in this work are positive for a first distributed implementation of a B&B algorithm using a high-productivity language. On its best results, Chapel-BB is can be slightly faster or equivalent to MPI-PBB on 32 locales (1024 cores). Moreover, it is up to 30% slower for other 2 instances. It is worth to mention that the MPI-PBB implementation is a state-of-the-art algorithm, and our experience with MPI+X is much higher than with Chapel. Moreover, we did not try to mimic MPI-PBB. Those results were achieved by only using the Chapel's productivity aware features for distributed programming, and the programmings models implemented by Chapel were followed.

The load balancing is the segment of code for which MPI-PBB dedicates

more SLOC compared to Chapel-BB. As a consequence, speedups higher than 60% of the linear one can be observed for 78% of the test cases. In contrast, for Chapel-BB, this value drops for 40% of the test-cases. The poor scalability faced by Chapel-BB for the smallest instances shows that there is room for improvement. A future research direction is to incorporate into Chapel components of the work stealing present in MPI-PBB. We do not plan to use lower level features, such as MPI or ZeroMQ library for implementing load balancing and communication. Instead, the objective is to harness the built-in features provided by Chapel for exploiting locality. As Chapel is an open-source language, the produced load balancing could be incorporated into the language as a distributed iterator.

6.3. Road Towards Exascale

One of the major obstacles on the road to the exascale computing era is dealing efficiently with scalability up to millions of cores. Another observation that can be made from the last editions of Top500 is that the cores are mostly supplied in low-energy computing resources (GPU, MIC, etc.). Therefore, dealing with scalability implicitly induces the heterogeneity issue [43]. According to Chapel's official documentation, the Xeon Phi accelerator is supported. However, there is no information concerning the support of GPUs. Another critical issue the scalability comes with is fault tolerance, because harnessing millions of cores results in a very high probability of failure [44]. The presence of fault tolerance in Chapel would encourage the adoption of this language for programming ultra-scale optimization algorithms. There is a work that proposes resilience features for Chapel [45]. However, these features were not incorporated into the language.

6.4. Main Insights

The following summarizes the main insights from our study on the use of Chapel for programming distributed B&B search algorithms.

- Researchers familiar with shared memory programming can incrementally design a distributed B&B algorithm using Chapel.
- It is possible to achieve performance and scalability competitive to MPI+ Pthreads using a high-productivity language.
- The productivity-aware features for load balancing provided by Chapel are not enough for efficiently exploiting the computer resources of several locales in more irregular scenarios.
- The C-interoperability features of Chapel are crucial for productivity. It was possible to reuse legacy code and focus on other aspects of the program, rather than porting the bounding functions to Chapel.
- The support for GPU and fault tolerance are crucial features that would encourage the use of Chapel for programming large-scale parallel optimization algorithms.

7. Conclusions and Future Works

In this paper, we have investigated the use of Chapel high-productivity language for the design and implementation of distributed B&B algorithms for solving combinatorial optimization problems. The proposed algorithm was implemented using the productivity-aware features of Chapel for distributed programming and applies both the global-view of control flow and PGAS programming models, instead of the well-known SPMD.

According to the productivity-oriented results, Chapel presents an overall expressiveness almost 6× higher, and it is up to 8× more productive than MPI+Pthreads. Researchers familiar with shared memory programming can incrementally design a distributed B&B algorithm using Chapel. Despite the high level of its features for distributed programming, it is possible to hand-optimize the data structures involved in the search process, and also incorporate legacy code written in C, which is essential in the context of exact parallel optimization.

Concerning performance and scalability, the best results of Chapel are equivalent to MPI+Pthreads on 32 locales (1024 cores). However, B&B algorithms are usually highly irregular applications, and the overhead of distributing the PGAS data structure, allied to the difficulties faced by its built-in distributed load balancing strategies, limits the scalability of Chapel-BB.

Concerning the adoption of Chapel, it is worth pointing out that users may be reluctant to learn another language [15]. The capacity of Chapel to include C code can be a partial solution for this situation. One could use C along with Chapel's high-productivity features for distributed programming. Moreover, C/C++ code usually exploits one library for each level of parallelism. Using Chapel along with C-interoperability may represent an equivalent learning curve. Finally, fault tolerance and the support of accelerators such as GPUs are crucial features towards exascale that are currently missing in Chapel.

As future work, we plan to program a work stealing technique that could be incorporated into the language as a distributed iterator. We also plan to run experiments on large-scale clusters. This way, it would be possible to investigate the limits of the productivity-aware features of Chapel concerning performance and scalability. Another future work is to compare Chapel to other PGAS-based libraries and high-productivity languages, such as OpenSHMEM, UPC, Dash, and Julia.

Acknowledgments

The experiments presented in this paper were carried out on the Grid'5000 testbed [46], hosted by INRIA and including several other organizations [3]. We thank Bradford Chamberlain, Elliot Ronaghan (Cray inc.) and Paul Hargrove (Berkeley lab.) for helping us to run GASNet on GRID5000. We also thank Michael Ferguson, Lydia Duncan (Cray inc.), and Louis Jenkins (University of Rochester) for their support on Chapel's Gitter platform [4].

[3] http://www.grid5000.fr
[4] https://gitter.im/chapel-lang/chapel

References

[1] W. Zhang, Branch-and-Bound Search Algorithms and Their Computational Complexity, Technical Report, DTIC Document, 1996.

[2] A. Y. Grama, V. Kumar, A survey of parallel search algorithms for discrete optimization problems, ORSA Journal on Computing 7 (1993).

[3] T. Crainic, B. Le Cun, C. Roucairol, Parallel branch-and-bound algorithms, Parallel combinatorial optimization (2006) 1–28.

[4] M. Mezmaz, N. Melab, E.-G. Talbi, A grid-enabled branch and bound algorithm for solving challenging combinatorial optimization problems, in: IEEE International Parallel and Distributed Processing Symposium, 2007. IPDPS 2007., IEEE, 2007, pp. 1–9.

[5] M. Mezmaz, R. Leroy, N. Melab, D. Tuyttens, A multi-core parallel branch-and-bound algorithm using factorial number system, in: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014, 2014, pp. 1203–1212. URL: `https://doi.org/10.1109/IPDPS.2014.124`. doi:`10.1109/IPDPS.2014.124`.

[6] J. Gmys, M. Mezmaz, N. Melab, D. Tuyttens, IVM-based parallel branch-and-bound using hierarchical work stealing on multi-GPU systems, Concurrency and Computation: Practice and Experience 29 (2017) e4019.

[7] N. Melab, J. Gmys, M. Mezmaz, D. Tuyttens, Multi-core versus many-core computing for many-task branch-and-bound applied to big optimization problems, Future Generation Computer Systems 82 (2018) 472 – 481.

[8] K. Aida, T. Osumi, A case study in running a parallel branch and bound application on the grid, in: The 2005 Symposium on Applications and the Internet, IEEE, 2005, pp. 164–173.

[9] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams,

et al., The landscape of parallel computing research: A view from berkeley, Technical Report, Technical Report UCB/EECS-2006-183, EECS Department, University of , 2006.

[10] S. Fiore, M. Bakhouya, W. W. Smari, On the road to exascale: Advances in high performance computing and simulationsan overview and editorial, Future Generation Computer Systems 82 (2018) 450 – 458.

[11] P. San Segundo, C. Rossi, D. Rodriguez-Losada, Recent developments in bit-parallel algorithms, INTECH Open Access Publisher, 2008.

[12] F. Feinbube, B. Rabe, M. von Löwis, A. Polze, NQueens on CUDA: Optimization issues, in: Parallel and Distributed Computing (ISPDC), 2010 Ninth International Symposium on, IEEE, 2010, pp. 63–70.

[13] T. Carneiro Pessoa, J. Gmys, F. H. de Carvalho Junior, N. Melab, D. Tuyttens, GPU-accelerated backtracking using CUDA dynamic parallelism, Concurrency and Computation: Practice and Experience (2017) e4374–n/a.

[14] G. Da Costa, T. Fahringer, J. A. R. Gallego, I. Grasso, A. Hristov, H. D. Karatza, A. Lastovetsky, F. Marozzo, D. Petcu, G. L. Stavrinides, et al., Exascale machines require new programming paradigms and runtimes, Supercomputing frontiers and innovations 2 (2015) 6–27.

[15] E. Lusk, K. Yelick, Languages for high-productivity computing: the DARPA HPCS language project, Parallel Processing Letters 17 (2007) 89–102.

[16] B. L. Chamberlain, E. Ronaghan, B. Albrecht, L. Duncan, M. Ferguson, B. Harshbarger, D. Iten, D. Keaton, V. Litvinov, P. Sahabu, et al., Chapel comes of age: Making scalable programming productive, in: Cray User Group, 2018.

[17] D. Callahan, B. L. Chamberlain, H. P. Zima, The cascade high productivity language, in: Ninth International Workshop on High-Level Parallel

Programming Models and Supportive Environments, 2004. Proceedings., IEEE, 2004, pp. 52–60.

[18] Cray Inc., Chapel language specification v.986, Cray Inc. (2018).

[19] G. Almasi, PGAS (partitioned global address space) languages, in: Encyclopedia of Parallel Computing, Springer, 2011, pp. 1539–1545.

[20] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, A. Navarro, User-defined parallel zippered iterators in chapel, in: Proceedings of Fifth Conference on Partitioned Global Address Space Programming Models, 2011, pp. 1–11.

[21] N. Dun, K. Taura, An empirical performance study of chapel programming language, in: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IEEE, 2012, pp. 497–506.

[22] B. L. Chamberlain, S.-E. Choi, M. Dumler, T. Hildebrandt, D. Iten, V. Litvinov, G. Titus, The state of the chapel union, Proceedings of the Cray User Group (2013) 114.

[23] C. Blum, A. Roli, Metaheuristics in combinatorial optimization: Overview and conceptual comparison, ACM Computing Surveys (CSUR) 35 (2003) 268–308.

[24] G. J. Woeginger, Exact algorithms for NP-hard problems: A survey, in: Combinatorial optimization – eureka, you shrink!, Springer, 2003, pp. 185–207.

[25] E. Alba, E.-G. Talbi, G. Luque, N. Melab, Metaheuristics and parallelism, Parallel Metaheuristics: A New Class of Algorithms (2005) 79–103.

[26] C. Papadimitriou, K. Steiglitz, Combinatorial optimization: algorithms and complexity, Dover Pubns, 1998.

[27] M. R. Garey, D. S. Johnson, R. Sethi, The Complexity of Flowshop and Jobshop Scheduling, Mathematics of Operations Research 1 (1976) pp. 117–129.

[28] B. J. Lageweg, J. K. Lenstra, A. H. G. R. Kan, A general bounding scheme for the permutation flow-shop problem, Operations Research 26 (1978) 53–67.

[29] S. M. Johnson, Optimal two- and three-stage production schedules with setup times included, Naval Research Logistics Quarterly 1 (1954) 61–68.

[30] T. Carneiro, F. H. de Carvalho Júnior, N. G. P. B. Arruda, A. B. Pinheiro, Um levantamento na literatura sobre a resolução de problemas de otimização combinatória através do uso de aceleradores gráficos, in: Proceedings of the XXXV Ibero-Latin American Congress on Computational Methods in Engineering (CILAMCE), Fortaleza-CE, Brasil, 2014, pp. 1–20.

[31] S. Tschoke, R. Lubling, B. Monien, Solving the traveling salesman problem with a distributed branch-and-bound algorithm on a 1024 processor network, in: 9th International Parallel Processing Symposium, 1995. Proceedings., IEEE, 1995, pp. 182–189.

[32] R. Machado, S. Abreu, D. Diaz, Parallel local search: Experiments with a PGAS-based programming model, CoRR abs/1301.7699 (2013).

[33] D. Munera, D. Diaz, S. Abreu, P. Codognet, A parametric framework for cooperative parallel local search, in: Evolutionary Computation in Combinatorial Optimization - 14th European Conference, EvoCOP 2014, Granada, Spain, April 23-25, 2014, Revised Selected Papers, 2014, pp. 13–24.

[34] D. Grünewald, C. Simmendinger, The GASPI API specification and its implementation GPI 2.0, in: 7th International Conference on PGAS Programming Models, 2013, p. 243.

[35] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, V. Sarkar, X10: An object-oriented approach to non-uniform cluster computing, ACM Sigplan Notices 40 (2005) 519–538.

[36] J. Gmys, Heterogeneous cluster computing for many-task exact optimization - Application to permutation problems, Theses, Université de Mons (UMONS) ; Université de Lille, 2017. URL: https://hal.inria.fr/tel-01652000.

[37] E. Taillard, Benchmarks for basic scheduling problems, European journal of operational research 64 (1993) 278–285.

[38] G. Karypis, V. Kumar, Unstructured tree search on SIMD parallel computers, IEEE Transactions on Parallel and Distributed Systems 5 (1994) 1057–1072.

[39] T. C. Pessoa, J. Gmys, N. Melab, F. H. de Carvalho Junior, D. Tuyttens, A gpu-based backtracking algorithm for permutation combinatorial problems, in: International Conference on Algorithms and Architectures for Parallel Processing, Springer, 2016, pp. 310–324.

[40] M. Snir, D. A. Bader, A framework for measuring supercomputer productivity, The International Journal of High Performance Computing Applications 18 (2004) 417–432.

[41] J. Kepner, HPC productivity: An overarching view, The International Journal of High Performance Computing Applications 18 (2004) 393–397.

[42] K. Kennedy, C. Koelbel, R. Schreiber, Defining and measuring the productivity of programming languages, The International Journal of High Performance Computing Applications 18 (2004) 441–448.

[43] M. J. Schulte, M. Ignatowski, G. H. Loh, B. M. Beckmann, W. C. Brantley, S. Gurumurthi, N. Jayasena, I. Paul, S. K. Reinhardt, G. Rodgers, Achieving exascale capabilities through heterogeneous computing, IEEE Micro 35 (2015) 26–36.

[44] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, et al., Addressing failures

in exascale computing, The International Journal of High Performance Computing Applications 28 (2014) 129–173.

[45] K. Panagiotopoulou, H.-W. Loidl, Towards resilient chapel: Design and implementation of a transparent resilience mechanism for chapel, in: Proceedings of the 3rd International Conference on Exascale Applications and Software, University of Edinburgh, 2015, pp. 86–91.

[46] R. Bolze, F. Cappello, E. Caron, M. Dayde, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quétier, O. Richard, E.-G. Talbi, I. Touche, Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed, International Journal of High Performance Computing Applications 20 (2006) 481–494.